1. **True/False.** Circle the appropriate choice (there are no trick questions).

(a) **T** In C, a pointer variable is used for storing an address.

(b) **F** In x86 assembly, the eax register can be used to hold integers but not addresses.

(c) **T** In x86 assembly, the instruction “shl eax,3” (or “shl $3,%eax” in AT&T syntax) multiplies the value in eax by 8.

(d) **F** If, in C, an array is declared by “int a[10];”, then accessing `a[11]` during execution will generate an error message.

(e) **T** A callee-saved register should be saved before it is written to in a procedure.

(f) **T** C has no built-in boolean type, rather the value 0 is used to represent false and all other values represent true.

(g) **F** ebp is a caller-saved register.

(h) **F** In x86 assembly, local variables in a procedure are generally declared in the .data section.

(i) **F** Compiling C into x86 assembly comprises translating each C statement into a single assembly instruction.

(j) **T** In x86 assembly, a label corresponds to an address.

2. **Answer this question on this sheet**

Consider the following x86 code fragment for computing the sum of an array of 10 32-bit integers, pointed to by ecx.

```
# Intel Syntax
mov eax,0
mov edx,0

TOP:
    cmp edx,10
    jl OUT
    add eax,[ecx+edx]
    inc edx
    jmp TOP

# AT&T Syntax
mov $0,%eax
mov $0,%edx

TOP:
    cmp $10,%edx
    jl OUT
    add (%eax,%edx,4),%eax
    inc %edx
    jmp TOP

OUT:
```

(a) There are two bugs in the code. What are they?

   The conditional jump should be a “jge” instruction and each array element should be accessed as “[eax+edx*4]” or “(%ecx,%edx,4)”.

(b) Put a comment following each “#” in the above code to describe what that instruction does (or is supposed to do, in the case of a bug).

   See the code above.

4. Put your answers in the blue book. Given the following C procedure

```c
int sum(int *a, int num)
{
    int i;
    int sum = 0;
    for(i=0; i<num; i++)
        sum = sum + a[i];
    return sum;
}
```

using the appropriate x86 addressing modes, fill in the missing code in the assembly procedure below that corresponds to the procedure `sum()`, above. Choose either Intel or AT&T syntax.

**Answer:** The missing code is shown below, underlined.

```
sum:
    push ebp
    mov ebp,esp
    push ebx
    mov ecx,[ebp+8] # ecx holds a
    mov edx,[ebp+12] # edx holds num
    mov eax,0 # eax holds sum
    mov ebx,0 # ebx holds i

    TOP:
    cmp ebx,edx #compare i to num
    jge OUT #jump out if i >= num
    add eax,[ecx+ebx*4] #add a[i] to sum
    inc ebx #increment i
    jmp TOP #jump to top of loop

    OUT:
    pop ebx #restore ebx
    pop ebp
    ret
```

5. Put your answer in the blue book.

(a) Define a C struct type `CELL` that contains the following fields: an integer `x`, a string `y`, and a `next` field that points to another structure of type `CELL`.

**Answer:**

```c
typedef struct cell {
    int x;
    char *y;
```
struct cell *next;
} CELL;

(b) Write in C a procedure corresponding to the declaration

int list_length(CELL *head);

that returns the length of a linked list whose first element is pointed to by head.

Answer:

int list_length(CELL *head)
{
    int count = 0;
    CELL *p = head;
    while (p != NULL) {
        count++;
        p = p->next;
    }
}


A simple C procedure for computing the integer square root of a number (i.e. the greatest
integer less than or equal to the square root of the number) is shown below (where “>>” is
the shift-right operator):

int square_rt(int x, int low, int high)
{
    if (low >= high - 1)
        return low;
    int mid = (low + high) >> 1;
    if ((mid * mid) > x)
        return square_rt(x,low,mid);
    else
        return square_rt(x,mid,high);
}

where the initial call to square_rt would be square_rt(n,1,n). Translate square_rt into
x86 assembly, so that it could be called from C.

Intel Syntax:

.globl _square_rt
_square_rt:
push ebp
mov ebp,esp

push ebx  #using ebx, so save it

mov eax,[ebp+12]  #eax gets low
mov ecx,[ebp+16]  #ecx gets high
sub ecx,1  #now ecx has high-1
cmp eax,ecx      # if low >= high-1
jge DONE         # eax already contains low, just return

mov ecx,[ebp+16] #reload ecx with high
mov edx,eax      #edx gets mid, first get low
add edx,ecx      #add high
shr edx          #shift right by one

mov ebx,edx      #ebx will hold mid*mid
imul ebx,ebx     #if mid*mid > x
jg GREATER       # jump to GREATER

#otherwise call square_rt(x,mid,high)
push ecx         #push high
push edx         #push mid
push DWORD PTR [ebp+8]  #push x
call _square_rt
add esp,12       #adjust stack
jmp DONE         #we're done, result is already in eax

GREATER:

#calling square_rt(x,low, mid)
push edx         #push mid
push eax         #push low
push DWORD PTR [ebp+8]  #push x
call _square_rt
add esp,12       #adjust stack
wec're done, result is already in eax

DONE:

pop ebx          #restore ebx
pop ebp
ret

AT&T syntax:

.globl _square_rt

_square_rt:
push %ebp
mov %esp,%ebp
push %ebx      #using ebx, so save it

mov 12(%ebp),%eax  #eax gets low
mov 16(%ebp),%ecx  #ecx gets high
sub $1,%ecx      #now ecx has high-1
cmp %ecx,%eax    # if low >= high
jge DONE        # eax already contains low, just return
mov 16(%ebp),%ecx  # reload ecx with high
mov %eax,%edx    # edx gets mid, first get low
add %ecx,%edx    # add high
shr %edx         # shift right by one
mov %edx,%ebx    # ebx will hold mid*mid
imul %ebx,%ebx   # ebx will hold mid*mid
cmp 8(%ebp),%ebx # if mid*mid > x
jg GREATER       # jump to GREATER
push %ecx        # push high
push %edx        # push mid
pushl 8(%ebp)    # push x
call _square_rt  # otherwise call square_rt(x,mid,high)
add $12,%esp     # adjust stack
jmp DONE         # were done, result is already in %eax

GREATER:

push %edx        # push mid
push %eax        # push low
pushl 8(%ebp)    # push x
call _square_rt  # calling square_rt(x,low,mid)
add $12,%esp     # adjust stack
jmp DONE         # result is already in %eax

DONE:

pop %ebx         # restore ebx
pop %ebp